Desafio de Criptografia Simétrica: Implementação do DES e Mecanismo de Troca de Chaves Diffie-Hellman

Angêlica Rita de Araújo[1] **, Ananda Guedes do Ó** [2]

[11] Instituto Federal da Paraíba (IFPB) – João Pessoa - PB – Brasil.
[2] Instituto Federal da Paraíba (IFPB) – João Pessoa - PB – Brasil.

Abstract. *This article presents the implementation of the DES (Data Encryption Standard) algorithm and the Diffie-Hellman key exchange mechanism using the Python programming language. The objective is to explore how this encryption algorithm functions in conjunction with the keys provided by the mechanism, within a client/server structure. To ensure message transmission, this process will use socket connections via the TCP/IP protocol and a predefined port number. Upon establishing the connection, private and public keys are generated, along with a shared key that will be used for message encryption and decryption.*

Resumo. *Este artigo apresenta a implementação do algoritmo DES (Data Encryption Standard) e o mecanismo de troca de chaves Diffie-Hellman utilizando a linguagem Python, com o intuito de explorar o funcionamento desse algoritmo de criptografia em conjunto com as chaves que serão providas do mecanismo, a partir de uma estrutura de cliente/servidor. De modo a garantir a transmissão de mensagens, esse processo irá utilizar a conexão via sockets pelo protocolo TCP/IP e um número de porta definido. Ao estabelecer a conexão, ocorre a geração de chaves privadas e públicas, bem como uma chave compartilhada que será utilizada para a criptografia da mensagem e sua descriptografia.*

1. Informações gerais

Este artigo tem como objetivo resolver e documentar um desafio proposto pela disciplina de Segurança de Dados no curso de Sistemas para Internet do Instituto Federal da Paraíba. O mesmo busca proporcionar uma experiência prática na implementação do Data Encryption Standard (DES) e do algoritmo de troca de chaves Diffie-Hellman utilizando a linguagem Python sem uso de bibliotecas auxiliares.

2. Fundamentos teóricos

2.1. Data Encryption Standard

O Data Encryption Standard (DES) foi desenvolvido em 1975 pelos pesquisadores da IBM, liderados por Horst Feistel, Don Coppersmith e outros, em colaboração com a National Security Agency (NSA) dos Estados Unidos. O algoritmo foi baseado no projeto original conhecido como Lucifer[1], criado na IBM e padronizado pelo National Institute of Standards and Technology (NIST, então chamado NBS). Ele utiliza uma estrutura de ciframento em bloco empregando a rede de Feistel como base teórica. A cifra de Feistel é um modelo para cifras de bloco baseado na ideia de cifras de produto, que aplicam

[1]UFRGS. *DES – Data Encryption Standard.* Disponível em: `http://penta2.ufrgs.br/gere96/segur2/des.htm`. Acesso em: 2 fev. 2025.

múltiplas cifras simples em sequência para aumentar a segurança. Ela permite criar uma cifra de bloco eficiente com um número menor de transformações possíveis do que uma cifra de bloco ideal.

A estrutura proposta por Feistel segue o conceito de substituições e permutações, em que na **substituição** elementos do texto claro são substituídos por elementos do texto cifrado. Na **permutação** a ordem dos elementos do texto claro é reorganizada sem alteração de valores. Esta estrutura opera dividindo o bloco de entrada em duas metades e passando-as por múltiplas rodadas de transformação. Cada rodada utiliza uma subchave derivada da chave principal e aplica uma função F à metade direita, combinando seu resultado com a metade esquerda por meio de uma operação XOR. Depois, as metades são trocadas, garantindo maior difusão e confusão dos dados. O número de rodadas, o tamanho do bloco e da chave, o algoritmo de geração das subchaves e a função F são parâmetros críticos para a segurança do sistema (STALLINGS, 2014).

O DES implementa essa estrutura usando blocos de 64 bits, chaves de 56 bits e 16 rodadas, tornando-se um dos exemplos mais conhecidos da aplicação do modelo de Feistel. Ele equilibra segurança e eficiência, embora suas características tenham sido superadas por algoritmos modernos. Desta forma, o DES foi amplamente adotado como padrão de criptografia pelo governo dos EUA em 1977, mas foi substituído devido à sua vulnerabilidade frente à crescente capacidade computacional para ataques de força bruta.

2.2. Diffie-Hellman

Stallings (2014, p. 226) afirma que ”o primeiro algoritmo de chave pública apareceu no artigo inicial de Diffie e Hellman que definia a criptografia de chave pública, e geralmente é chamado de troca de chaves Diffie-Hellman. Diversos produtos comerciais empregam essa técnica de troca de chaves”. Ele é um protocolo que permite que duas partes, sem conhecimento prévio uma da outra, estabeleçam conjuntamente uma chave secreta compartilhada sobre um canal de comunicação público e inseguro. Ele foi introduzido em 1976 por Whitfield Diffie e Martin Hellman, pesquisadores da Universidade de Stanford.

A segurança do algoritmo baseia-se no problema matemático do logaritmo discreto, que é considerado computacionalmente difícil de resolver em grupos finitos, como os grupos multiplicativos de inteiros módulo um número primo. Ele utiliza propriedades de exponenciação modular, garantindo que a troca de informações públicas não revele a chave privada (MENEZES, OORSCHOT, VANSTONE; p. 515, 1996). Conforme os próprios autores (DIFFIE-HELLMAN, 1976), a função matemática utilizada segue a seguinte lógica:

Parâmetros públicos: Dois valores são conhecidos publicamente: um número primo q e uma raiz primitiva a de q.

Geração das chaves públicas: O usuário A escolhe um número aleatório privado X_A, com $0 < X_A < q$, e calcula a chave pública Y_A como: $Y_A = a^{X_A} q$. O usuário B escolhe um número aleatório privado X_B, com $0 < X_B < q$, e calcula a chave pública Y_B como: $Y_B = a^{X_B} q$.

Cálculo da chave secreta compartilhada: O usuário A calcula a chave secreta K como: $K_A = Y_B^{X_A} q$. O usuário B calcula a chave secreta K como: $K_B = Y_A^{X_B} q$. Ambos os usuários, A e B, chegam ao mesmo valor para a chave secreta compartilhada,

$K_A = K_B$, devido às propriedades da aritmética modular.

A segurança do algoritmo baseia-se na dificuldade de calcular logaritmos discretos. Embora a exponenciação modular seja fácil de calcular, o cálculo inverso (o logaritmo discreto) é considerado computacionalmente difícil, o que garante a segurança do processo. Isso se torna inviável para números grandes. Além disso, sua eficiência reside na dificuldade de ataques conhecidos, como força bruta ou cálculos diretos do logaritmo discreto, quando parâmetros adequados são escolhidos. O Diffie-Hellman é um marco na criptografia moderna, sendo a base para protocolos como o Transport Layer Security (TLS) e outros sistemas de criptografia de chave pública.

3. Metodologia

A metodologia implementada neste trabalho teve como objetivo a comunicação entre dois arquivos Python, sender.py e receive.py, utilizando o algoritmo de troca de chaves Diffie-Hellman para o estabelecimento de uma chave secreta compartilhada, e o algoritmo de criptografia simétrica DES para garantir a confidencialidade da mensagem. A seguir, são detalhados os métodos de ambos os arquivos e o fluxo de comunicação entre eles.

3.0.1. Estrutura Geral do Sistema

A comunicação entre os arquivos é estabelecida através de sockets TCP/IP, em que o arquivo sender.py atua como cliente e o receive.py como servidor. O processo de troca de mensagens envolve três etapas principais: a troca de chaves, utilizando a classe Diffie-Hellman presente no arquivo diffiehellman.py; a criptografia da mensagem, utilizando a função des_encrypt, cujo algoritmo está no arquivo des.py, e seus respectivos arrays para permutação e substituição no arquivo tables.py; e, por fim, a descriptografia da mensagem no arquivo receive.py, por meio da função des_decrypt.

3.0.2. Implementação da Troca de Chaves

A implementação apresentada em Python encapsula o processo de troca de chaves em uma classe chamada DiffieHellman, que permite a geração de chaves privadas e públicas, bem como o cálculo de uma chave secreta compartilhada. Esta classe foi construída para receber dois parâmetros essenciais para o funcionamento do protocolo, que são comuns para o receptor e o emissor:

- **prime_number**: um número primo grande, usado como módulo para as operações exponenciais.
- **base**: um número inteiro que atua como a base da exponenciação modular.

No construtor da classe (**init**), além de armazenar esses valores citados anteriormente, também são definidos dois atributos adicionais que serão preenchidos posteriormente:

- **secret_key**: um número aleatório secreto gerado para cada usuário.
- **public_key**: valor público calculado a partir da chave secreta.

O método generate_secret_key cria um número aleatório que servirá como a chave secreta do participante. Esse número é escolhido dentro do intervalo [2, prime_number -

2], garantindo que esteja dentro do domínio adequado para os cálculos modulares subsequentes. Após a geração da chave secreta, o método compute_public_key calcula a chave pública utilizando a exponenciação modular através do método pow, interno do Python, utilizando como parâmetros as variáveis base, secret_key e prime_number. Este mesmo método é utilizado na função compute_shared_key, utilizando a public_key, secret_key e o prime_number. No momento em que são instanciados os objetos da classe DiffieHellman nas variáveis sender e receive em seus respectivos arquivos, as funções descritas anteriormente são colocadas em uso.

3.0.3. Implementação do DES

O DES foi implementado no arquivo des.py, parte da implementação do algoritmo foi baseada nos passos descritos no livro *Introdução a criptografia* escrito por Marcelo Ferreira Zochio (2016). Com fins de organização, os arrays para permutação e substituição foram organizados no arquivo tables.py, quais sejam: ip_table, shift_schedule, e_table, p_table, s_box, pc1_table, pc2_table, ip_inverse_table. Estes são chamados no código de acordo com a necessidade lógica de cada função. Abaixo é possível observar uma tabela com todas as funções necessárias para operar o DES:

Função	**Descrição**
`hex_to_bin(hex_str)`	Converte uma string hexadecimal para uma string binária.
`bin_to_hex(bin_str)`	Converte uma string binária para uma string hexadecimal.
`bin_to_dec(binary)`	Converte um número binário para decimal.
`dec_to_bin(num)`	Converte um número decimal para binário, garantindo que tenha múltiplos de 4 bits.
`permute(block, table)`	Reorganiza os bits de um bloco de acordo com uma tabela de permutação.
`xor(a, b)`	Aplica a operação XOR entre duas strings binárias.
`left_shift(key, shifts)`	Realiza um deslocamento à esquerda em uma chave.
`generate_keys(key)`	Gera chaves de rodada a partir da chave inicial, utilizando permutações e deslocamentos.
`des_process_block (plaintext_hex, round_keys)`	Executa o algoritmo DES em um bloco, aplicando substituições, permutações e XOR com as chaves de rodada.
`format_key(key)`	Converte uma chave em string para uma representação binária de 64 bits.
`des_encrypt (plaintext_hex, key)`	Executa a criptografia DES em um bloco de texto usando a chave fornecida.
`des_decrypt (ciphertext_hex, key)`	Executa a descriptografia DES revertendo as rodadas de criptografia.

Table 1. Descrição das funções do algoritmo DES

Para entender o fluxo do algortimo, partimos da função principal, des_encrypt pois

a mesma inicia o processo de criptografia de um texto fornecido em formato hexadecimal, juntamente com a chave a ser utilizada. Destacamos que esta função é utilizada no sender.py. Ao receber estes parametros, a des_encrypt aciona outras funções:

- *format_key*: A chave fornecida é inicialmente formatada pela função format_key, pois esta é um número inteiro, produto do algoritmo de Diffie-Hellman. Sendo assim, ela é convertida de seu formato original para string, posteriornente para hexadecimal, e por fim, para uma representação binária de 64 bits. Se o comprimento da chave binária for menor que 64 bits, ela é preenchida com zeros à esquerda até atingir esse comprimento para atender os requisitos do algoritmo no momento de criar as subchaves.
- *generate_keys*: A chave binária formatada é então passada para a função generate_keys. Nesse processo, a chave original é permutada usando uma tabela fixa, chamada PC1_TABLE, que é fornecida no arquivo tables.py. O resultado dessa permutação é dividido em duas metades, chamadas de ”esquerda” e ”direita”. Cada uma dessas metades passa por deslocamentos à esquerda, definidos na tabela SHIFT_SCHEDULE. Após o deslocamento, as duas metades são combinadas novamente e passadas por outra permutação, chamada PC2_TABLE, para gerar uma chave de 56 bits para cada uma das 16 rodadas do DES. Essas chaves geradas são armazenadas em uma lista chamada round_keys, que será utilizada no processo de criptografia.
- *des_process_block*: Após a geração das chaves de rodada, a função des_process_block é chamada, recebendo o texto criptografado (em formato hexadecimal) e a lista de chaves de rodada.O texto hexadecimal (representando o bloco de dados a ser criptografado) é convertido para uma string binária através da função hex_to_bin.O bloco binário resultante é permutado de acordo com uma tabela chamada IP_TABLE, realizando uma reorganização dos bits. E então ele é dividido em duas metades de 32 bits: uma parte esquerda e uma parte direita. Essas metades passarão por transformações nas rodadas subsequentes. Em cada uma das 16 rodadas do DES, a metade direita do bloco é expandida para 48 bits usando a tabela E_TABLE. Essa expansão é seguida por uma operação XOR com a chave de rodada correspondente. O resultado dessa operação passa por uma substituição usando as tabelas S_BOX, gerando uma string binária de 32 bits, que é permutada usando a tabela P_TABLE. A metade esquerda do bloco passa então por uma operação XOR com o resultado da transformação. Em seguida, a metade esquerda e direita são trocadas de posição para a próxima rodada. Após as 16 rodadas de transformação, as duas metades do bloco (esquerda e direita) são novamente permutadas utilizando a tabela IP_INVERSE_TABLE. Esse processo retorna o bloco criptografado.
- *bin_to_hex*: O bloco final (agora criptografado) é convertido de volta para formato hexadecimal pela função bin_to_hex, que retorna o texto criptografado.

A função des_decrypt é usada para descriptografar a mensagem no arquivo receive.py. O fluxo e as funções usadas são semelhantes a função des_encrypt, pois, ela também realiza o processo de descriptografia de um texto cifrado representado em hexadecimal usando uma chave de entrada por meio da função format_key. Difere-se no seguinte ponto: na momento em que a função generate_keys é utilizada dentro da des_decrypt, foi necessário adicionar o operador [::-1] para inverter a ordem das chaves

da rodada, revertendo corretamente as transformações aplicadas durante a criptografia. As demais funções e operações são aplicadas de maneira semelhante.

3.0.4. Comunicação via Sockets

Os arquivos receive.py e sender.py são responsáveis por estabelecer a comunicação entre um servidor e um cliente utilizando sockets TCP, e colocar em prática o protocolo Diffie-Hellman para troca de chaves e o algoritmo DES para criptografia de mensagens por meio de suas respectivas funções.

- O **receive.py** (servidor) cria um socket na porta 12345, aguarda a conexão do cliente e inicia o protocolo Diffie-Hellman para gerar uma chave compartilhada segura. Após receber a chave pública do cliente, ele calcula a chave secreta comum e usa essa chave para descriptografar a mensagem recebida via DES.
- O **sender.py** (cliente) conecta-se ao servidor via socket e também executa o protocolo Diffie-Hellman, trocando chaves públicas e derivando a mesma chave compartilhada. Ele então criptografa uma mensagem fixa com DES e a envia ao servidor.

A comunicação via sockets permite que ambos os programas troquem dados de forma estruturada, enquanto a troca de chaves Diffie-Hellman garante que a chave de criptografia seja segura sem precisar ser transmitida diretamente.

4. Resultados

O algoritmo apresentou os resultados esperados em relação aos processos de encriptação, decriptação e troca de chaves. A comunicação foi estabelecida por meio da troca de mensagens cifradas, utilizando uma chave pública compartilhada para a criptografia das informações.

Ao inicializar o arquivo receive.py no terminal, o sistema entra em estado de espera pela conexão de um cliente na porta pré-definida. Em seguida, o arquivo sender.py é executado em outro terminal, estabelecendo a conexão com a porta designada, o que garante a comunicação entre os dois processos. Durante esse procedimento, ambos os arquivos utilizam números inteiros previamente acordados para gerar a chave pública, armazenando-a em uma variável por meio do protocolo de chaveamento Diffie-Hellman (Figura 1).

Dessa forma, os arquivos receive.py e sender.py compartilham suas chaves públicas e realizam o cálculo da chave compartilhada (Figura 2). Com o canal de comunicação estabelecido, o cliente envia uma mensagem criptografada utilizando o algoritmo implementado no arquivo des.py para o servidor (Figura 3). O servidor, por sua vez, aplica o mesmo algoritmo para realizar a decriptação da mensagem recebida (Figura 4). Após a conclusão da transmissão, os terminais são encerrados imediatamente por meio da finalização dos sockets de comunicação.

```
prime_number = 23
base = 432788601367
receive = DiffieHellman(prime_number, base)
secret_key = receive.generate_secret_key()
public_key = receive.compute_public_key()
print(f"Chave pública (receiver): {public_key}")
```

Figure 1. Criação das chaves públicas e privadas.

```
# Calcula a chave compartilhada
shared_key = sender.compute_shared_key(public_key_receive)
print(f"Chave compartilhada (usada no DES): {shared_key}")
```

Figure 2. Chave compartilhada que será usada no algoritmo DES.

```
mensagem = "0123456789ABCDDF"
msg_encrypt = des.des_encrypt(mensagem, shared_key)
```

Figure 3. Mensagem a ser encriptada e chamada da função de encriptação.

```
# Recebe a mensagem criptografada do sender
msg_encrypt = client_socket.recv(1024).decode()
print(f"Mensagem criptografada recebida: {msg_encrypt}")

# DES - Descriptografia
msg_decrypt = des.des_decrypt(msg_encrypt, shared_key)
print(f"Mensagem descriptografada: {msg_decrypt}")
```

Figure 4. Processo de recebimento e de descriptografia por parte do receive.

Exemplo 1:

- Para o primeiro exemplo utilizaram-se os valores de número primo = 23 e base = 432788601367. E os respectivos valores de chave encontrados foram 13, para a chave pública do receiver, 10, para a chave pública do sender e 2 para a chave compartilhada (Figura 5).
- A mensagem a ser criptografada foi ”0123456789ABCDDF” (Figura 2).
- O valor encontrado após a criptografia foi ”5D9782F4A14D28CF”, que ao ser criptografado retornou ao seu valor original (Figura 6).

```
Chave pública (receiver): 13
Chave pública (sender) recebida: 10
Chave compartilhada: 2
```

Figure 5. Valor das Chaves.

```
Mensagem criptografada recebida: 5D9782F4A14D28CF
Mensagem descriptografada: 0123456789ABCDDF
```

Figure 6. Mensagem recebida criptografada e descriptografada.

Exemplo 2:

- Para o segundo exemplo utilizaram-se os valores de número primo = 31 e base = 266463123968. E os respectivos valores de chave encontrados foram 2, para

a chave pública do receiver, 2, para a chave pública do sender e 8 para a chave compartilhada (Figura 7).
- A mensagem a ser criptografada foi ”0123456789ABCDDF” (Figura 2) .
- O valor encontrado após a criptografia foi ”5D9782F4A14D28CF”, que ao ser criptografado retornou ao seu valor original (Figura 8).

```
Chave pública (receiver): 2
Chave pública (sender) recebida: 2
Chave compartilhada: 8
```

Figure 7. Valor das Chaves.

```
Mensagem criptografada recebida: 3DF1C89225A9B614
Mensagem descriptografada: 7F2B9CDE1345A68B
```

Figure 8. Mensagem recebida criptografada e descriptografada.

5. Discussão

O algoritmo de troca de chaves Diffie-Hellman, implementado para gerar chaves compartilhadas entre o cliente e o servidor, foi eficaz no estabelecimento de uma comunicação segura. No processo de implementação, um desafio foi garantir a precisão dos cálculos das chaves públicas e secretas entre o receive.py e sender.py, uma vez que esses valores são fundamentais para a criptografia e decriptação das mensagens. Embora o algoritmo tenha funcionado corretamente, ele aplicado sozinho, sem a utilização do DES, apresenta vulnerabilidade, pois existe a possibilidade de ataques de *man-in-the-middle* se os dados não forem validados adequadamente, já que as chaves são trocadas publicamente e podem ser interceptadas.

A respeito da implementação do DES, foram enfrentados desafios significativos, especialmente em relação a manipulação dos bits nas funções de permutação e substituição, com a necessidade de adequar o tamanho dos dados em diferentes estágios (64 bits para a entrada e 56 bits para a chave), pois este é um processo complexo que aumenta a chance de erros, especialmente quando se lida com a geração das chaves e a conversão entre formatos binário, hexadecimal e string.

Seu uso sem adição de outros algoritmos de segurança de dados pode ser inseguro, pois o mesmo enfrenta limitações, como a vulnerabilidade a ataques de força bruta devido ao tamanho reduzido da chave (56 bits). Para mitigar essas limitações, uma alternativa, caso o objetivo seja continuar utilizando a lógica do DES, seria a utilização do 3DES, uma versão aprimorada do DES que aplica o algoritmo três vezes com diferentes chaves, proporcionando maior resistência a ataques. No entanto, o 3DES também enfrenta desafios, como o aumento do tempo de processamento e a complexidade na manipulação dos dados, dado que a entrada e a chave precisam ser corretamente ajustadas e o algoritmo exige três passagens de criptografia.

Outras melhorias poderiam incluir a substituição do DES pelo Advanced Encryption Standard (Padrão Avançado de Criptografia), sendo um algoritmo de criptografia simétrica utilizado mundialmente para proteger dados, que oferece maior segurança e eficiência.

6. Conclusão

O uso combinado do algoritmo DES e do protocolo Diffie-Hellman se mostrou uma solução eficaz para garantir a segurança em comunicações. O protocolo Diffie-Hellman possibilita a troca segura de chaves entre cliente e servidor, permitindo que ambos gerem uma chave compartilhada sem transmiti-la diretamente, o que reduz o risco de interceptação. A chave compartilhada gerada é então utilizada pelo DES para criptografar e descriptografar as mensagens trocadas, garantindo a confidencialidade dos dados. Essa combinação oferece uma camada robusta de proteção, pois, mesmo que um atacante intercepte a comunicação, ele não teria acesso à chave secreta ou às informações criptografadas.

Os testes realizados confirmaram que o sistema cumpre suas funções de troca de chaves, criptografia e descriptografia de maneira eficiente, como demonstrado pelo processo de inicialização e execução dos scripts receive.py e sender.py. No entanto, uma possível melhoria seria a substituição do DES por algoritmos de criptografia mais modernos, como o AES, que oferecem maior segurança contra ataques.

7. References

DIFFIE, W.; HELLMAN, M. New Directions in Cryptography. IEEE Transactions on Information Theory, v. 22, n. 6, p. 644-654, 1976.

MENEZES, A. J.; VAN OORSCHOT, P. C.; VANSTONE, S. A. Handbook of Applied Cryptography. Boca Raton: CRC Press, 1996.

STALLINGS, William. Criptografia e Segurança de Redes: Princípios e Práticas. 6. ed. São Paulo: Pearson, 2014.

ZOCHIO, Marcelo Ferreira. Introdução à criptografia. 1. ed. São Paulo: Editora NOVATEC, 2016.